Programación II, 2015-2016
Escuela Politécnica Superior, UAM

Práctica 2: TAD Pila

PARTE 2.A. PREGUNTAS SOBRE LA PRÁCTICA

Responded a las siguientes preguntas **completando el fichero disponible en el .zip**. Renombrad el fichero para que se corresponda con vuestro nº de grupo y pareja de prácticas y adjuntadlo al fichero .zip que entreguéis.

1. En el ejercicio 4 hemos realizado una serie de modificaciones para que la cima de la pila sea de tipo **puntero**. Indicad qué ficheros habéis tenido que modificar y cuáles han sido estos cambios:

En **elestack.h**, antes:	Ahora:
En **elestack.c**, antes:	Ahora:
En **stack.h,** antes:	Ahora:
En **stack.c,** antes:	Ahora:
En stack_ini, se inicializa la pila, con el valor del tope -1. En stack_free, se hace un bucle for que va liberando los elementos en la posición tope, hasta que este vale -1. Al meter o extraer un elemento en la pila, se incrementa o disminuye en uno el valor del tope, y se mete o extrae el elemento en la posición del tope. En stack_top, se hace una copia del elemento en la posición que vale tope y devuelve la copia. El bucle de stack_print, llama a elestack_print, pasando como argumento desde el valor del tope hasta 0.	En stack_ini, se inicializa la pila, con el tope apuntando a NULL. En stack_free, se hace un bucle while que va liberando los elementos mientras tope sea mayor o igual que la dirección del elemento 0. En stack_pop, hay que diferenciar si la pila tiene un elemento o tiene más. En el primer caso, se modifica top para que apunte a NULL. Si no, que apunte una dirección menor. De manera similar, en stack_push hay que diferenciar si la pila está vacía o no. En stack_top, se hace una copia del elemento en la dirección a la que apunta el tope y devuelve la copia. El bucle de stack_print, llama a elestack_print, pasando como argumento desde la dirección del tope hasta la dirección del elemento en la posición 0.
En **p2_e3.c**, antes:	Ahora:

2. Una aplicación habitual del TAD PILA es para evaluar expresiones posfijo. Describe en detalle qué TADs habría que definir/modificar para poder adaptar la pila de enteros (P2_E1) o de puntos (P2_E2) en una que permita evaluar expresiones posfijo.

   Habría que definir el TAD Operación donde estén definidos el símbolo que representa cada operación y que tenga una función que reciba un carácter como argumento y que devuelva si es un operando o un operador y otra función que reciba dos operadores y un operando y que devuelva el resultado de la operación.
   Los TADs Stack y EleStack, no hace falta modificarlos.
   Otra posibilidad es definir un nuevo TAD Cadena de Caracteres, e incluir las funciones mencionadas del TAD Operación. En ella se definiría el símbolo de fin de cadena y habría una función que leyera una cadena, a parte de las funciones usuales que se pueden hacer sobre una cadena de caracteres.

PARTE 2.B. MEMORIA SOBRE LA PRÁCTICA

Explica las decisiones de diseño y alternativas que se han considerado durante la práctica. En particular, explica en detalle la implementación y decisiones de diseño de los ejercicios **P2_E2** y **P2_E3**.

En el ejercicio dos de la práctica, se mantuvo el diseño del ejercicio uno, modificando únicamente la estructura de EleStack por un puntero a punto en vez de un puntero a entero. Esto obligó a cambiar los prototipos de las funciones (elestack-int.h). La implementación de elestack-int.c es similar al del ejercicio uno, pero donde antes eran enteros, ahora son puntos.

En el ejercicio tres discutimos diferentes alternativas para implementar el algoritmo que imprime el camino encontrado. Decidimos definir una nueva enumeración llamada Posible, porque era más intuitivo para la función recorrer, que devuelve POSIBLE si existe un camino, NO_POSIBLE si no existe y FALLO si ha habido algún error durante la ejecución. Para recorrer el laberinto teníamos que etiquetar cada punto como descubierto si se ha llegado hasta él desde el punto de entrada y explorado si se habían descubierto los puntos vecinos. La opción que descartamos fue definir dos símbolos nuevos como descubierto y explorado, porque ello requería modificar el símbolo del punto, que se perdería, y modificaría el laberinto. Con el objetivo de mantener los símbolos intactos, definimos una nueva enumeración llamado Estado. En la estructura del punto se añadió un dato de tipo Estado que se inicializa a Nada al crear el punto y luego lo etiqueta como descubierto y explorado. Aunque esta opción requiere incluir dos nuevas funciones en point.c que son point_getEstado y point_setEstado y sus prototipos en point.h.
Para encontrar el camino, había que ir almacenando el padre de cada punto. Para ello, podíamos haber añadido un puntero a punto en la estructura Point que fuese el padre o haber añadido dos enteros que fuesen la coordenada x e y del padre. La primera alternativa requería añadir solo dos funciones (point_getPadre y point_setPadre) pero complicaba más el código de point_copy y de point_equals, dado que se podían acceder a estas funciones sin que los puntos tuviesen padre. Así que optamos por la segunda, que requería añadir cuatro funciones (point_getCoordxPadre, point_getCoordyPadre, point_setCoordxPadre y point_setCoordyPadre) pero que simplifican mucho más el código.

CONCLUSIONES FINALES

Estos cuatro ejercicios nos han sido muy útiles para poner en práctica la teoría sobre pilas vista en clase. El tercer problema fue el más complicado, porque había que hacer muchos cambios con respecto al ejercicio anterior. Implementar el pseudocódigo de la función que recorre un laberinto y que devuelve si se puede llegar o no hasta la salida, no fue demasiado difícil. Pero dar con un algoritmo que encontrase un posible camino, si lo había, nos costó más. Después, llegamos a una idea general que era asociar a cada punto un "padre" desde el que había sido descubierto, pero había distintas alternativas para resolverlo. Además, este algoritmo obligaba a añadir algunas funciones en el TAD Point y modificar la estructura del punto. Una vez resuelto el ejercicio tres, no tardamos mucho en hacer el cuarto. Había que cambiar algunos códigos de las funciones de stack.c y este nos sirvió para asentar los conocimientos sobre punteros. En cuanto al primer ejercicio, le dedicamos bastante tiempo porque iba a ser la base de los tres siguientes. Tuvimos algunas dificultades con stack_print, porque no nos imprimía correctamente la pila. Pero nos dimos cuenta de que el bucle que imprime los elementos no estaba bien. Finalmente, el ejercicio dos fue el que más rápido hicimos y no tuvimos ninguna dificultad.